Gemini CLI: The Operator's Manual

A Guide to Wielding a Command-Line Agent

Table of Contents

- Introduction: The Prompt is a Stage
- Part 1: The Mic Check - Basic Operations
  - 1.1: The Core Loop - The Rhythm of Interaction
  - 1.2: Casing the Joint - Reading the Environment
  - 1.3: The First Edit - Creating and Deleting
  - 1.4: The Shell Game - Running System Commands
  - 1.5: Best Practices - How to Think
- Part 2: The Flow State - Complex Maneuvers
  - 2.1: The Search - Finding the Needle
  - 2.2: The Surgical Strike - Precision File Edits
  - 2.3: Context is King - The Art of the Follow-up
  - 2.4: Chaining Commands - Building a Combo
- Part 3: The Architect - Building Worlds
  - 3.1: The Blank Canvas - Scaffolding New Projects
  - 3.2: The Refactor - Rewriting the Old Code
  - 3.3: The Test-Driven Hit - Writing and Running Tests
  - 3.4: The Automation Script - Building Your Own Agent
- Glossary: The Lingo
- Index

Introduction: The Prompt is a Stage

Forget what you know about chatbots. This isn't a conversation. It's a collaboration. This tool, this Gemini CLI, isn't here to talk about code; it's here to write it, run it, and fix it. It's a command line with a brain, an agent at your fingertips.

Most tools give you a hammer and expect you to find the nail. This is different. It's a multi-tool that reconfigures itself based on your intent. You don't need to know the exact syntax for every

little thing. You need to know how to state your mission clearly. The gap between your idea and its execution just got smaller.

This guide isn't a list of commands. It's a manual on how to operate. How to think. How to direct. From asking it to find a file to telling it to build a full application. The prompt is your stage. Let's learn how to run the show.

Part 1: The Mic Check - Basic Operations

First, you learn the fundamentals. The rhythm. The basic moves that every other complex operation is built on. Get this down, and you're ready for anything.

1.1: The Core Loop - The Rhythm of Interaction

This is the basic beat: `You Ask -> I Act -> You See`.

1. **You Ask:** You state your goal in plain English. "List the files here." "Read the content of `README.md`."
2. **I Act:** I translate your goal into a specific tool call, like `list_directory` or `read_file`. You'll see this proposed action. You give the green light.
3. **You See:** The tool runs. The output appears. Now you have new information to make your next move.

Mastering this loop is everything. It's a back-and-forth, a rally. Your job is to keep the rally going until the mission is done.

1.2: Casing the Joint - Reading the Environment

You can't operate in a place you can't see. Before you change anything, you get the lay of the land.

- **`list_directory`:** This is you opening your eyes. Use it to see what files and folders are in a directory. Always start here. *"Show me what's in the `src` directory."*

- **`read_file`:** Once you spot a file that looks interesting, you read it. You don't guess what's inside. You look. *"Read `config.json`."*
- **`read_many_files`:** When you need the bigger picture. You don't read one email; you read the whole thread. This tool lets you grab the content of multiple files at once. *"Read all the `.js` files in the `src/components` directory."*

1.3: The First Edit - Creating and Deleting

Reading is passive. Now, we make a mark.

- **`write_file`:** The most direct tool. You want a file to exist with specific content? This is your weapon. You can create a new file or completely overwrite an old one. It's a clean slate. *"Create a file named `hello.txt` with the content 'Hello, World!'."*
- **`run_shell_command` with `rm`:** This is how you take things off the board. There's no dedicated `delete_file` tool because the shell already has the best one: `rm`. *"Delete the file `hello.txt`."* I will translate this to `run_shell_command(command='rm hello.txt')`. Respect the power here; there is no undo.

1.4: The Shell Game - Running System Commands

Sometimes, the built-in tools aren't the right weapon for the job. Sometimes you need raw power. That's when you use **`run_shell_command`**. It's your escape hatch to the full power of the underlying system shell (`bash`, `zsh`, etc.).

- **Use it for:** Installing packages (`npm install`), running builds (`npm run build`), initializing projects (`git init`), or any other system command I don't have a dedicated tool for.
- **The Catch:** This is the heavy artillery. I will tell you what command I'm about to run and what it does. You

have to give the go-ahead. If you tell me to do something reckless, I'll do it, but the responsibility is yours.

1.5: Best Practices - How to Think

- **Be Direct, Not Polite:** This isn't a conversation. It's a command. Get to the point. "Please could you possibly list the files for me?" is slow. "List files in `src`." is fast.
- **One Mission at a Time:** Start with one clear goal. Don't say, "Read the file and then change the first line and then tell me if it works." Say, "Read the file `main.py`." Then, after you see it, say, "Now, replace this specific line with this new one."
- **I Have No Memory of Your Files:** I don't "remember" what was in a file after I've read it. Every interaction is fresh. If you want me to edit a file, I need to read it first, *in that same interaction*, to get the exact context for the edit.

Part 2: The Flow State - Complex Maneuvers

Now the rhythm is second nature. It's time to start chaining moves together, to flow from one action to the next with speed and precision.

2.1: The Search - Finding the Needle

Your project is a haystack. These tools find the needle.

- **`glob`:** You know a piece of the filename, but not the exact path. `glob` is your search dog. It finds files matching a pattern. *"Find all files ending in `.css` inside the `src` directory."* -> `glob(pattern='src/**/*.css')`.
- **`search_file_content`:** You don't know the file, but you know a piece of the code inside it. This is your X-ray. It searches the *content* of files for a specific string or pattern. This is your most powerful tool for

discovery. *"Where is the function `calculateTotal` defined?"* -> `search_file_content(pattern='function calculateTotal')`.

2.2: The Surgical Strike - Precision File Edits

`write_file` is a sledgehammer. **`replace`** is a scalpel. It is designed for changing a specific piece of an existing file. To use it effectively, you must be precise.

1. **First, Read:** Always use `read_file` to get the exact, literal text you want to change.
2. **Provide Context:** When you tell me to replace something, don't just give me the one line. Give me the line *and* a few lines before and after it. This ensures I target the exact spot you intend and not a similar line somewhere else.
3. **State the Replacement:** Provide the new block of code that will replace the old one.

Bad prompt: "Change the title in `index.html`." *Good prompt:* "First, read `index.html`. (I read it). Okay, now in `index.html`, replace `<h1>Old Title</h1>` with `<h1>New, Better Title</h1>`."

2.3: Context is King - The Art of the Follow-up

I am not a script. I am an agent that maintains context *within a single session.* You can have a conversation.

- **Start Broad, Then Narrow Down:**
  1. *"Find all the configuration files in this project."* (I use `glob` to find `*.json`, `*.yaml`, etc.)
  2. *"Okay, read the contents of `config.prod.json` and `config.dev.json`."* (I use `read_many_files`.)
  3. *"In `config.prod.json`, change the database host to `prod.db.internal`."* (I use `replace`.)

This is the flow. You gather information, make a decision, and then give a precise command to act on that decision.

2.4: Chaining Commands - Building a Combo

While you give me one command at a time, you should be thinking three moves ahead. The output of one tool is the input for your next command.

The Pattern: `Discover -> Analyze -> Act`

- **Discover:** Find the relevant files. (`glob`, `search_file_content`)
- **Analyze:** Read the content of those files to understand the situation. (`read_file`, `read_many_files`)
- **Act:** Modify the files or run a command. (`replace`, `write_file`, `run_shell_command`)

This isn't just a sequence; it's a strategy. Internalize it.

Part 3: The Architect - Building Worlds

This is the final level. You stop thinking about editing files and start thinking about building systems. You don't give me tasks; you give me missions.

3.1: The Blank Canvas - Scaffolding New Projects

Don't build a new project by hand. Command it into existence.

Your Prompt: "Create a new React project called `my-app` using `create-react-app`. Then, create a new component file at `src/components/Header.js`. Finally, replace the default content of `App.js` with a basic component that imports this new `Header`."

My Execution Plan: 1. `run_shell_command(command='npx create-react-app my-app')` to build the project skeleton. 2. `write_file(file_path='my-app/src/components/Header.js', content='...')` to create the new component. 3.

`read_file(absolute_path='my-app/src/App.js')` to get the content of the default App. 4. `replace(...)` to swap out the default App content with your specified new code.

You laid out the grand vision. I executed the tactical steps.

3.2: The Refactor - Rewriting the Old Code

A refactor is not a series of small edits. It's a mission with a clear goal.

Your Prompt: "The file `src/utils.js` uses `var` everywhere. I want to refactor it to use modern JavaScript. Read the file, then rewrite the entire file to replace all instances of `var` with either `let` or `const`, based on whether the variable is reassigned or not."

My Execution Plan: 1. `read_file(absolute_path='src/utils.js')` to get the full context. 2. Analyze the code in my internal model to understand the variable assignments. 3. Construct the *entire new file content* in memory. 4. `write_file(file_path='src/utils.js', content='...')` to overwrite the old file with the new, refactored code.

3.3: The Test-Driven Hit - Writing and Running Tests

Don't just write code. Write code that works, and prove it.

Your Prompt: "I have a function in `math.js` called `add(a, b)`. I need you to create a new test file `math.test.js` that uses the Jest testing framework. The test should assert that `add(2, 2)` returns 4. After you create the test, run the test suite."

My Execution Plan: 1. `write_file(file_path='math.test.js', content='...')` with the appropriate Jest test code. 2. `run_shell_command(command='npm test')` to execute the test runner. 3. Report the output of the test run back to you.

3.4: The Automation Script - Building Your Own Agent

This is the final step. You use me to write a script that *you* can then run from your own terminal. You build your own tools.

Your Prompt: "Write me a bash script called `deploy.sh`. This script should take one argument, which is the commit message. It needs to run `npm run build`, then `git add .`, then `git commit -m "$1"`, and finally `git push`. Make the script executable."

My Execution Plan: 1. `write_file(file_path='deploy.sh', content='...')` with the bash script content. 2. `run_shell_command(comman` `+x deploy.sh')` to make it executable.

Now, the power is in your hands. You have used the agent to build a tool, graduating from operator to architect.

Glossary: The Lingo

- **Agent:** What I am. A tool that can reason, plan, and act to accomplish a goal.
- **Context:** The information I have *right now*. It comes from the current conversation and the output of recent tools. It is not a permanent memory.
- **Glob:** A pattern used to find files, like a wildcard. `*.js` finds all JavaScript files.
- **Prompt:** Your command. Your mission. The clearer the mission, the better the result.
- **Refactor:** Restructuring existing computer code without changing its external behavior.
- **Scaffolding:** Creating the basic skeleton of a new project or feature.
- **Shell:** The underlying command-line interpreter of the operating system (e.g., bash, zsh, PowerShell).
- **Tool Call:** The specific, structured command I execute based on your prompt (e.g., `read_file(...)`).

Index

- Automation: 6
- Best Practices: 4
- Context: 5
- Creating Files: 4
- Deleting Files: 4
- `glob`: 5
- `list_directory`: 4
- `read_file`: 4
- `read_many_files`: 4
- Refactoring: 6
- `replace`: 5
- `run_shell_command`: 4
- Scaffolding: 6
- `search_file_content`: 5
- Testing: 6
- `write_file`: 4